第4章

UMLとSystemCによるシステムLSI開発を体験する

——段階的に詳細化したDMAコントローラのSystemCモデルを評価

塚田雄一

ここでは，UMLとSystemCを利用するシステムLSIの開発工程のうち，SystemCの詳細化の過程を体験する．SystemCのトランザクション・レベル（TLM）のモデルを順次，バス・サイクル精度（BCA）レベル，レジスタ転送レベル（RTL）に置き換えていく．サンプルとなる回路は簡易DMA（direct memory access）コントローラである．SystemCモデリング・ツール/デバッガの評価版と各段階のモデル・データを，本誌付属のCD-ROMに収録した．ここではシミュレーションを行いながら，それぞれの段階のモデルの動きをチェックしていく．（編集部）

システムLSIの回路規模は年々増大し，また複雑化しています．現在，すでに1チップ当たり1,000万ゲートを搭載する時代になっていますが，1億ゲートの大台に乗るのもまもなくのことでしょう．こうなると，従来の開発手法ではいくつかの問題が出てきます．例えば，ひとりの設計者が開発できる回路規模には上限があります．設計の記述量（例えば，ハードウェア記述言語のコード量など）を減らすために，今後，設計抽象度の引き上げが求められると予想されます．

こうした動きは，ソフトウェア開発の歴史をひも解くとわかりやすいと思います．ソフトウェア開発では，「設計規模が増大→抽象度の引き上げ→さらに設計規模が増大→さらに抽象度の引き上げ」といったことを繰り返し，最近では，UML（unified modeling language）などのモデル・ベース設計手法が使われるようになっています．システムLSIの開発においても同様の変化が訪れると，筆者らは考えています．フレッシャーズのみなさんが設計現場の主力として活躍するころには，Verilog HDLやVHDLといったハードウェア記述言語に加えて，UMLやSystemC，C/

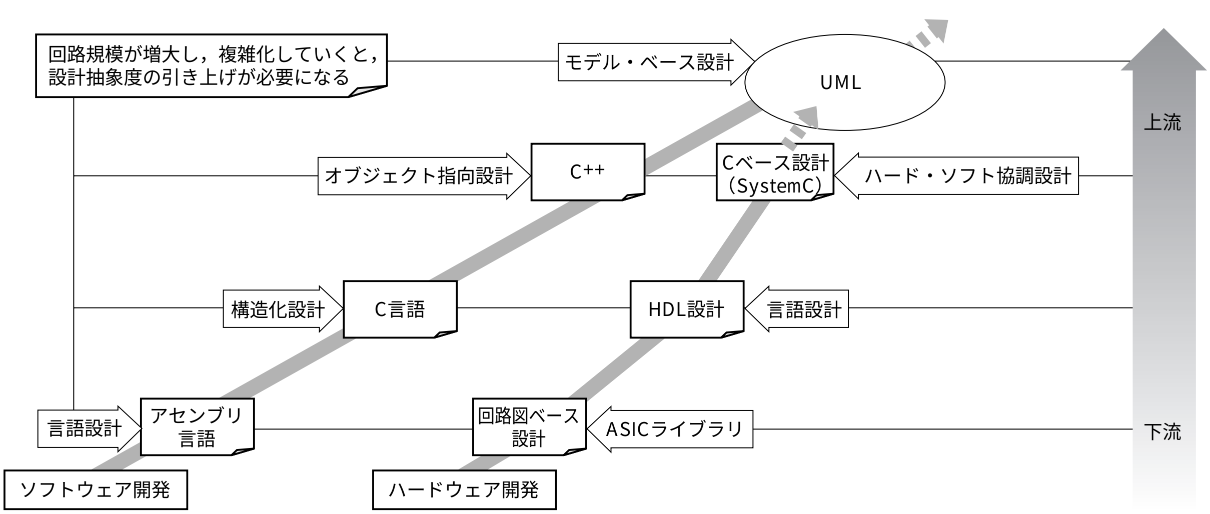


図1　ソフトウェア設計とハードウェア設計の歴史

ソフトウェアの表現形式は，「アセンブリ言語→C言語→C++→UML」と変遷していった．システムLSIの表現形式も，今後はC++をベースとするSystemCやUMLに変わっていくだろう．

C++ といった言語を駆使して大規模なシステムLSI を開発しているかもしれません（**図1**）．

現状では，ハードウェア（LSI）とソフトウェアは別々の技術者が別々の考えかたに基づいて開発しています．しかし，今後はハードウェアとソフトウェアの境界線はなくなり，両方を包括的に取り扱う「ハードウェア・ソフトウェア協調設計」や「システム・レベル設計」が重要になってくると思われます．そのため，ハードウェア技術者であっても，ソフトウェア向けの開発言語や開発環境を使いこなすスキルが求められるようになります．

こうした状況を踏まえて，本稿では，フレッシャーズの技術者を対象に，UML とSystemC によるシステムLSI 開発手法について具体的に解説していきます．

● TLM，BCA，RTL の３段階で詳細化

筆者ら（キャッツとリコー）は，本誌2004 年12 月号，pp.88-96 に「LSI をUML とSystemC でモデリングする」という記事を執筆しました．この記事では，制御用ASIC（application specific integrated circuit）のかなめであるDMA コントローラをUML でモデリングし，SystemC の構造図を作成する事例を紹介しました．

今回は，このSystemC の構造図をベースに，トランザクション・レベル・モデリング（TLM ： transaction level modeling）の表現から，バス・サイクル精度（BCA ： bus cycle accurate）レベルの表現，レジスタ転送レベル（RTL ：register transfer level）の表現へと詳細化していく過程を解説します（**図2**）．ここで，トランザクション・レベルとは，パケットやCPU 命令など，任意のデータ列（トランザクション）を一つの単位として取り扱いながらシステムを考える抽象度です．バス・サイクル精度は，バス・サイクルやクロック同期の概念を含む抽象度です．レジスタ転送レベルは，フリップフロップ（レジスタ）と組み合わせ回路によって機能を表現する抽象度です．

こうしたさまざまな設計抽象度の混在が可能である点は，SystemC の特徴の一つとなっています．

● モデリング・ツールとデバッガを使用

ここでは，SystemC モデリング・ツール「XModelink SoC Modeler（以下，SoC Modeler と呼ぶ）」とSystemC デバッガ「XModelink SystemC Debugger（以下，SystemC Debugger と呼ぶ）」の評価版を使いながら，SystemC を使った設計手法について解説していきます（**図3**）．これらのツールは本誌付属のCD-ROM に収録されています．

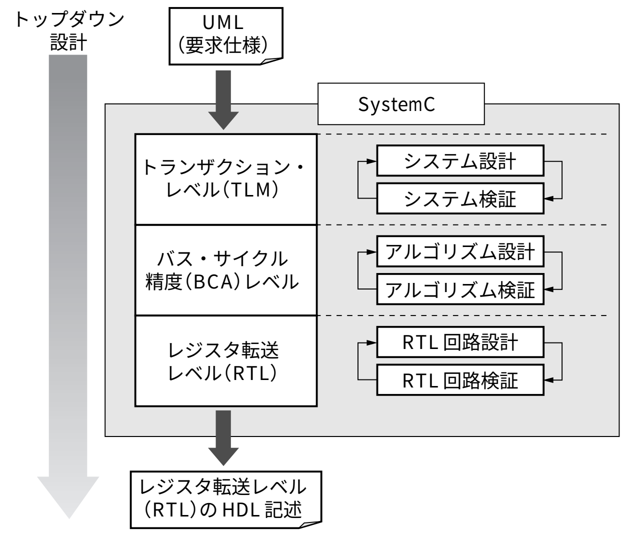


図2　SystemC 言語を用いたトップダウン設計
トップダウン設計では，トランザクション・レベルの簡素なモデルを徐々に具体化し（抽象度を下げ），レジスタ転送レベルの詳細なモデルへと書き換えていく．

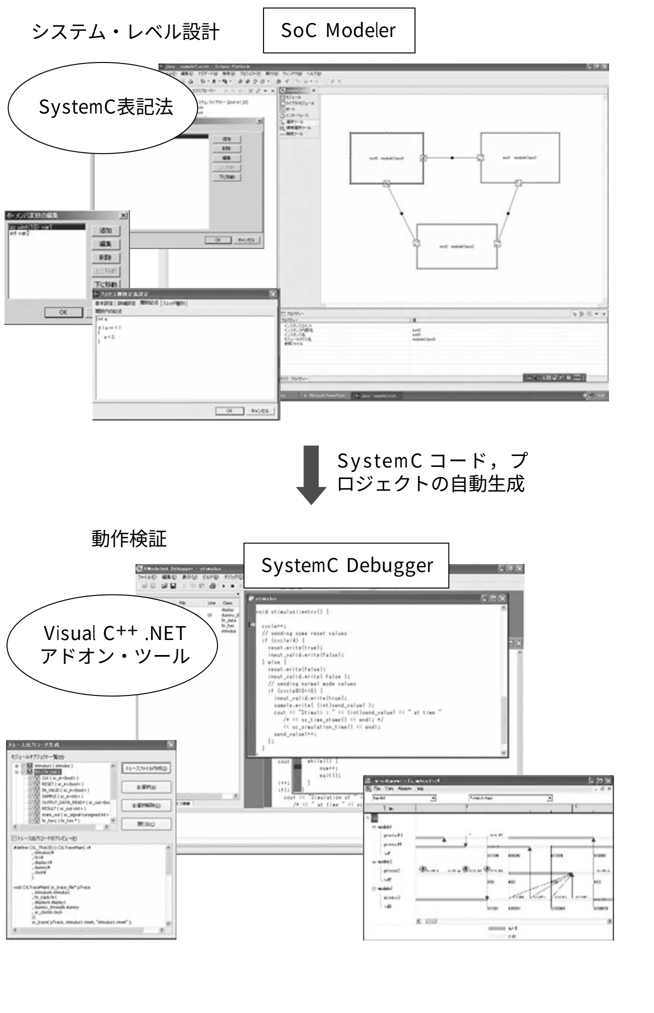
SoC Modeler は，米国IBM 社のUML モデリング・ツール「Rational Rose」で作成したUML のクラス図を読み込んで，SystemC に変換する機能を備えています．ユーザは，SystemC のグラフィカルな表記法（構造図）を使いながら，システムをモデリングしていきます．そのため，SystemC の文法の細部まで理解していなくても，正しいSystemC 記述を作成できます．また，ポート未接続のチェック機能なども備えています．

SystemC Debugger は，SystemC のソース・コード・デバッガです．ソース・コード・デバッガではありますが，ハードウェアの設計では必須である信号波形の表示機能も備えています．なお，本デバッガを使用する際には，パソコンに米国Microsoft 社の「Visual C++ .NET 2003」がインストールされている必要があります．

● DMA コントローラのサンプル・モデルを評価

今回は，サンプルとしてDMA コントローラのモデルを使用します．ただし，実際のシステムLSI の設計では，UML やSystemC を使ってDMA コントローラだけをモデリングするということはほとんどありません（もっと複雑な機能をモデリングするケースが多い）．また，今回設計したDMA コントローラについて，シミュレーションによ

4



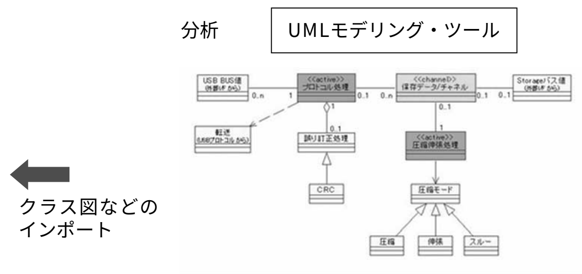


図3　XModelinkの構成

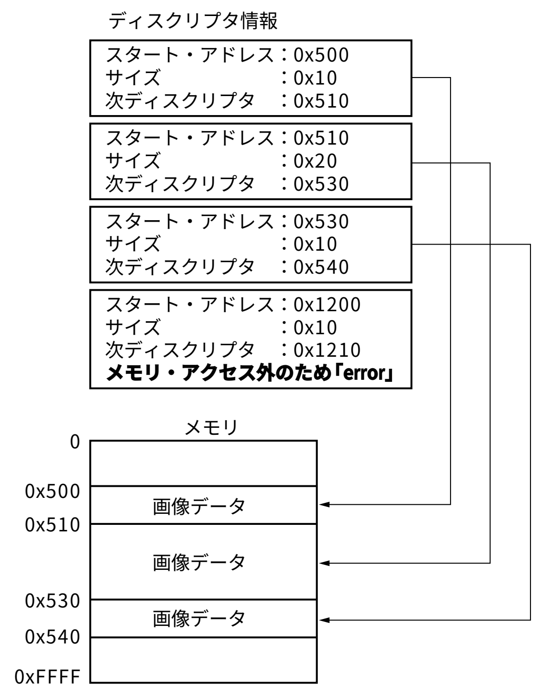
XModelinkは，UMLモデリング・ツールからクラス図をインポートし，SystemC表記法を使って設計していく「SoC Modeler」と，ソース・コード・デバッグと波形観測の機能を備える「SystemC Debugger」の二つのツールから構成されている．



(**a**) 転送イメージ

1. 画像処理ユニットとメモリの間でデータ転送を行う
2. データ転送方法は，ディスクリプタ情報に従う
3. 転送するデータは，2次元データのみ
4. データ転送終了後は，割り込み通知を行う
5. メモリ領域外にアクセスした場合は，エラー割り込み通知を行う

(**b**) 要求仕様の抜粋



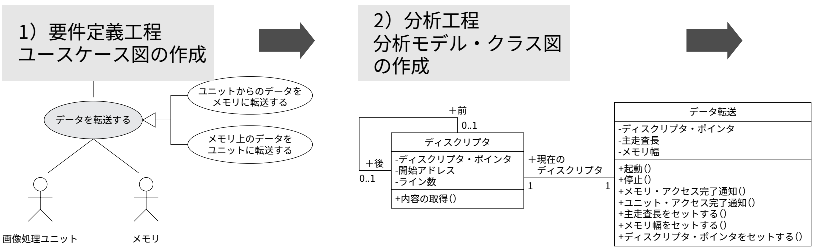
(**c**) ディスクリプタ情報とメモリ

図4　サンプル・システムの概要

本稿で使用するサンプル・システムの転送イメージ，要求仕様，ディスクリプタ情報を示した．

る機能検証は行っていますが，実際にLSI化して動作を確認したわけではありません．本サンプル・モデルは，あくまでも，フレッシャーズのみなさんに設計手法を理解していただくための例題と考えてください．

システムの要求仕様としては，画像処理ユニットとメモリの間でデータ転送を行う状況を考えます．また，データ転送方法は，ディスクリプタ情報（画像ユニット・スタート・アドレス，画像データ・サイズ，次のディスクリプタ・アドレス）に従うものとします．今回使用するディスクリプタ情報については，処理や検証を簡単にするために，データ転送サイズを減らしてあります．また，データの転送方向は「メモリ→画像処理ユニット」の単方向のみとしま



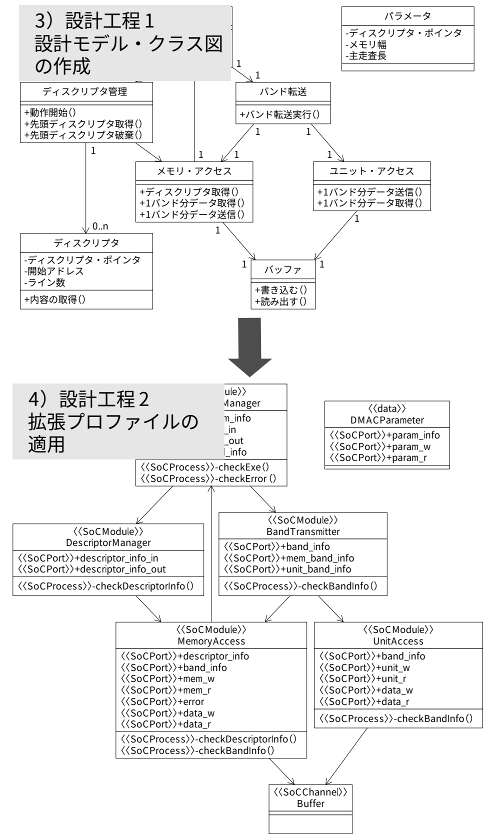


図5　UMLモデリングの例
要件定義工程（ユースケース図の作成），分析工程（分析モデル・クラス図の作成），設計工程1（設計モデル・クラス図の作成），設計工程2（拡張プロファイルの適用）の順に作業を進めていく．

す（**図4**）[注1]．

● UMLモデリングは4工程

SystemCで設計する前工程として，筆者らはUMLを使ってDMAコントローラをモデリングしました．このときの手順を簡単に紹介します（**図5**）．なお，UMLによるモデリングの詳細は本誌2004年12月号，pp.88-96の記事で示されているので，そちらを参照してください．また，UMLそのものについては，多くの教科書や解説書が市販されています．これらの書籍にもきちんと目を通しておいてください．

1）要件定義工程（ユースケース図の作成）――要求（要件）を定義するユースケース図を作成する．これにより，「DMAコントローラは何をするものなのか」を明確にする．

2）分析工程（分析モデル・クラス図の作成）――クラスの抽出とクラス間の協調関係を示す分析モデル・クラス図を作成する．要求を分析し，「実現のためには何が必要か」を検討する．

3）設計工程1（設計モデル・クラス図の作成）――分析モデル・クラス図の詳細化を行い，設計モデル・クラス図を作成する．機能の詳細を洗い出し，クラスごとに機能を割り当てる．

4）設計工程2（拡張プロファイルの適用）――ハードウェア合成（ビヘイビア合成）を意識してモデリングする．UML for SoCフォーラム[注2]が定義した拡張プロファイルを適用する．

● ライセンスを取得してツールをインストール

さっそくSystemCによるモデリングに入りたいところですが，その前に前述の二つのツールを手元のパソコンにインストールする必要があります．

まず，ツールの開発元であるキャッツに評価用ライセンスの発行を依頼します．会社名，部署名，名まえを明記し，件名（Subject）に「Design Wave 4月号用，1ヵ月ライセンス発行依頼」と入力して，info@zipc.comあてに電子メールを送ります．

評価用ライセンスを取得したら，二つのツールのインストールを開始します．それぞれのツールのディレクトリのinstall.exeを実行すると，インストールが開始されます．本誌付属のCD-ROMにインストール・マニュアルが収録されているので，細かい手順については，そちらを参照してください．

1点，注意していただきたいことがあります．SystemC DebuggerはVisual C++ .NET 2003のアドオン・ツールです．使用する際には，Visual C++ .NET 2003をあらか

注1：この仕様は，本誌2004年12月号，pp.88-96の記事で紹介したモデル（仕様）と多少異なる．
注2：UMLとSystemCを使ったLSIのシステム・レベル設計手法を策定し，普及推進を図る団体．2003年4月にキャッツ，日本ラショナルソフトウェア（現 日本アイ・ビー・エム），富士通の3社が共同で設立した．その後，NEC，キヤノン，東芝，メタボリックス，リコーが参加した．SystemCモデルとUMLモデルの対応を明確に定義したUML拡張プロファイル「UML Profile for SoC」を策定した．

じめインストールしておく必要があります．

● サンプル・モデルを読み込む

ツールをインストールできたら，あらかじめ用意されているサンプル・モデルを読み込みます．

まず，SoC Modelerを起動して，初期設定を行います．Windowsのスタート・メニューから「プログラム」→「XModelink SoC Modeler Ver1.2」→「XModelink SoC Modeler Ver1.2」を選択し，SoC Modelerを起動してください．

次に，インポート機能を用いて，サンプル・モデルを取り込みます．ナビゲータ領域でマウスを右クリックし，ポップアップ・メニューから「インポート」を選択してください．表示されたダイアログの「既存プロジェクトをワークスペースへ」を選択し，SoC Modelerのインストール・ディレクトリの中の「sample」を指定します．例えばインストール・ディレクトリが「C:\Program Files\CATS\SoC ModelerVer1.2」の場合，「C:\Program Files\CATS\SoC ModelerVer1.2\sample」を選択します．その後，［終了］ボタンを押すと，サンプル・モデルがインポートされます（**図6**）．

sampleプロジェクト・ツリー内の「tlm_sample.scstr」の部分をダブルクリックすると，ライセンス・キーの入力が求められます．開発元から入手したライセンス・キーを入力し，［OK］ボタンを押してください．ライセンス・キーの確認は，1度行えば以降は行う必要はありません．

● 6段階でモデルを詳細化

インポートしたサンプル・モデルの概要について説明していきます．

サンプル・モデルに含まれるモジュール（機能）はTransmissionManager，BandManager，DescriptorManeger，MemoryAccess，UnitAccessの五つです．それぞれのモジュールに対応するプロセス（処理）として，TransmissionManagerにはcheckExeとcheckErrorの二つが，BandManagerにはcheckBandInfoが，DescriptorManegerにはcheckDescriptorInfoが，MemoryAccessにはcheckDescriptorInfoとcheckBandInfoの二つが，UnitAccessにはcheckBandInfoがあります．つまり，サンプル・モデル全体では七つのプロセスが存在するのです．

ここでは，以下の六つのステップに分けて，段階的にモデルを詳細化していきます（**表1**）．それぞれのステップのモデルが，サンプル・モデルの中に含まれています．

ステップ1——本来はUMLモデリング・ツールからインポートされるモデル．トランザクション・レベルで表現されている．プロセスの記述にはSystemCのTHREADプロセスを使用する．

ステップ2——メモリと画像処理ユニットの間の転送部分をバス・サイクル精度モデルに変更する．checkBandInfoをCTHREADプロセスにする．

ステップ3——ディスクリプタ間をバス・サイクル精度モデルに変更する．checkDescriptorInfoをCTHREADプロセスにする．

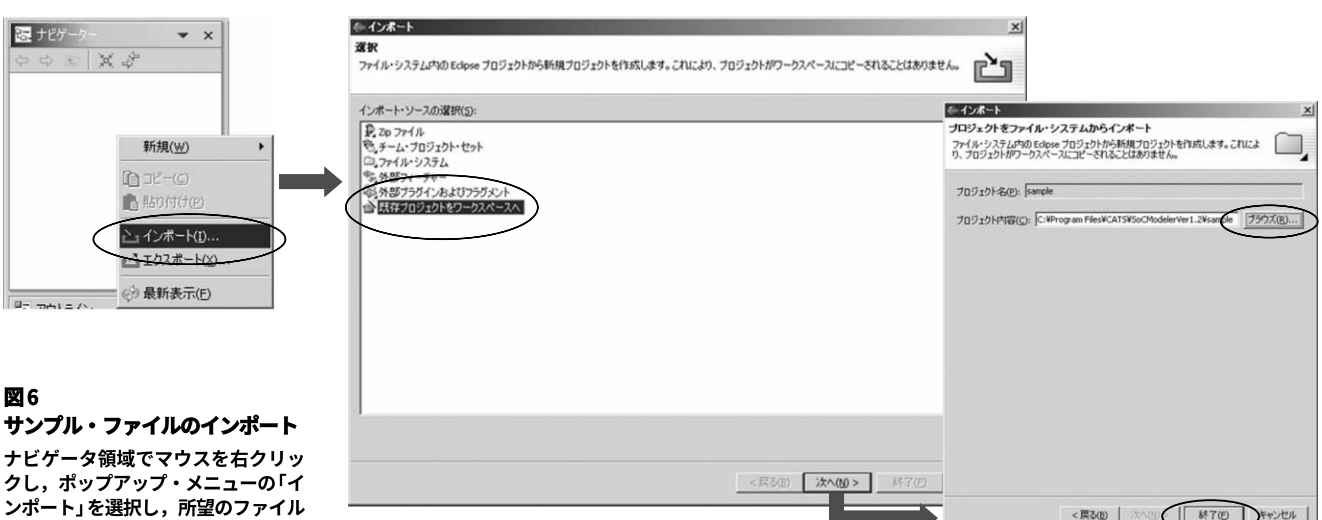


図6
サンプル・ファイルのインポート
ナビゲータ領域でマウスを右クリックし，ポップアップ・メニューの「インポート」を選択し，所望のファイルを指定する．

表1 プロセスの変化

機 能	TransmissionManager	BandManager	DescriptorManeger	MemoryAccess		UnitAccess
処 理	checkExe, checkError	checkBandInfo	checkDescriptorInfo	checkDescriptorInfo	checkBandInfo	checkBandInfo
ステップ1	THREAD	THREAD	THREAD	THREAD	THREAD	THREAD
ステップ2	↓	↓	↓	↓	CTHREAD	CTHREAD
ステップ3	↓	↓	CTHREAD	CTHREAD	↓	↓
ステップ4	CTHREAD	CTHREAD	↓	↓	↓	↓
ステップ5	↓	↓	↓	METHOD	METHOD	METHOD
ステップ6	METHOD	METHOD	METHOD	↓	↓	↓

ステップ4——すべてをバス・サイクル精度のレベルで表現する．残りの部分をCTHREADプロセスにする．
ステップ5——メモリと画像処理ユニットの間の転送部分をRTLモデルに変更する．checkBandInfo，checkDescriptorInfoをMETHODプロセスにする．
ステップ6——すべてをRTLで表現する．残りの部分をMETHODプロセスにする．

それぞれのステップで作成するモデルの詳細について説明していきます．

1）最初はトランザクション・レベルから

UMLモデリング・ツールからインポートされたモデルは，トランザクション・レベルと呼ばれる抽象度で表現されています（**図7(a)**）．このレベルでは，モジュールがどのような処理を行うかだけを記述します．また，プロセスの記述にはTHREADプロセスを使用します．モジュール間のデータの受け渡しについては，任意のデータ列（トランザクション）を一つの単位として考えます．ここではFIFO（first-in first-out：先入れ先出し）の概念を利用して，データ列は別のモジュールから入ってきた順番に転送されるものとします．つまり，トランザクション・レベルでは，「どのようなタイミング（クロック・サイクル）で，どのようなプロトコルを用いて転送する」といった細かい記述を行う必要がありません．

2）メモリ-ユニット間通信をバス・サイクル精度に

ステップ2では，メモリから画像処理ユニットへの通信をバス・サイクル精度レベルのモデルに置き換えます（**図7(b)**）．バス・サイクル精度レベルでは，プロセスの記述にはCTHREADプロセスを使用します．また，1クロックの処理ごとにwait()関数を挿入する必要があります（1クロックの処理を明示する）．各モジュール間のデータの受け渡しについても，1クロックごとの受け渡しの方法（プロトコル）を記述する必要があります．

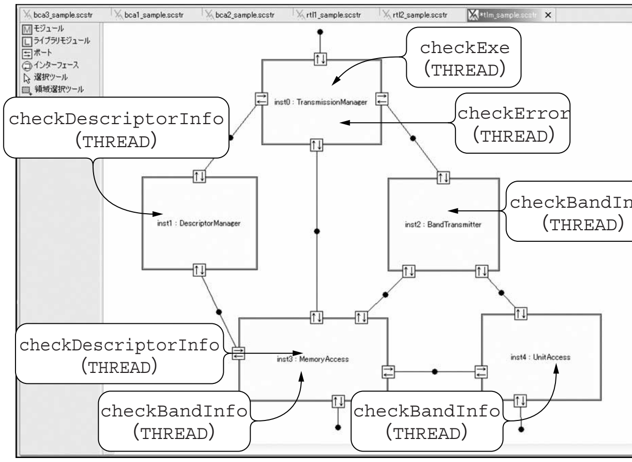
ここでは，メモリから画像処理ユニットへの転送部分についてのみ，バス・サイクル精度のレベルで記述しています．また，メモリ管理と画像処理ユニット管理の間のインターフェースに，通信開始のための「start」，およびデータ転送に使用する「data_r」を追加しました．さらに，各プロセスはクロック同期で動作するため，クロック「clk」を追加してあります．

帯域幅情報（BandInfo）がある場合，MemorryAccessモジュール内のcheckBandInfoプロセスでは，画像処理ユニット管理に対して5回データ転送を行うように記述してあります．また，UnitAccessモジュール内のcheckBandInfoプロセスについては，メモリからデータを受信し，画像処理ユニットにデータを書き込む処理を記述しています．

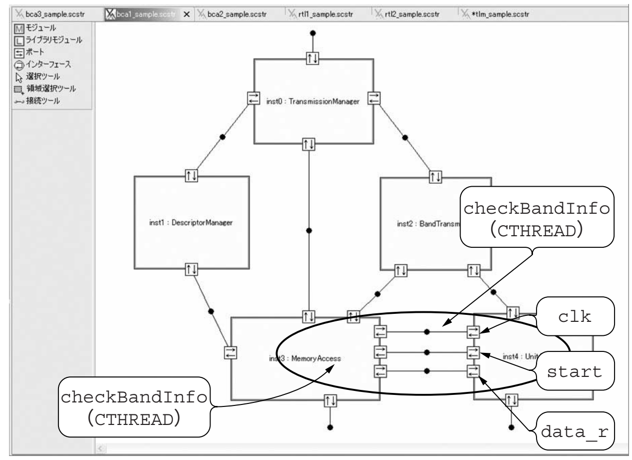
3）ディスクリプタ間通信をバス・サイクル精度に

ステップ3では，ディスクリプタ間をバス・サイクル精度レベルのモデルに置き換えます（**図7(c)**）．ディスクリプタとメモリ管理の間のインターフェースに，通信開始のタイミングを示す「memory_start」，メモリ転送アドレスである「address」，メモリの転送サイズを示す「size」を追加しました．また，DescriptorManegerモジュール内のcheckDescriptorInfoプロセスは，内部にディスクリプタ情報（番号）を持っています．TransmissionManagerモジュールからディスクリプタ情報を受信したときは，MemoryAccessモジュールに対してアドレスやサイズなどの情報を転送します．MemoryAccessモジュール内のcheckDiscriptorInfoプロセスでは，受信したアドレスやサイズがメモリの規定範囲内であるかどうかのチェックを行います．これがNGの場合はTransmissionManagerモジュールに対してエラー（error）を送り，OKの場合は内部データとして保持します．

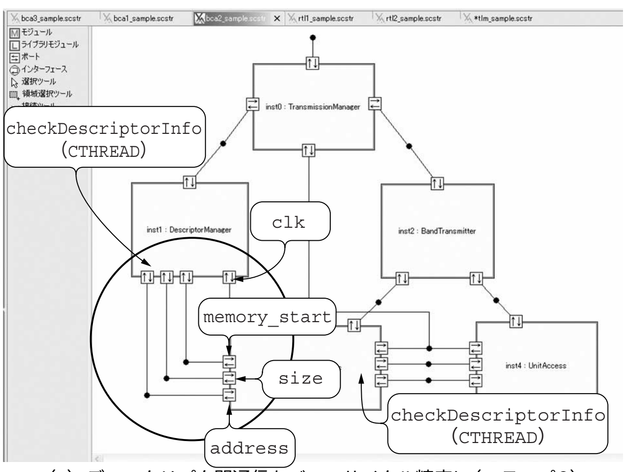
4



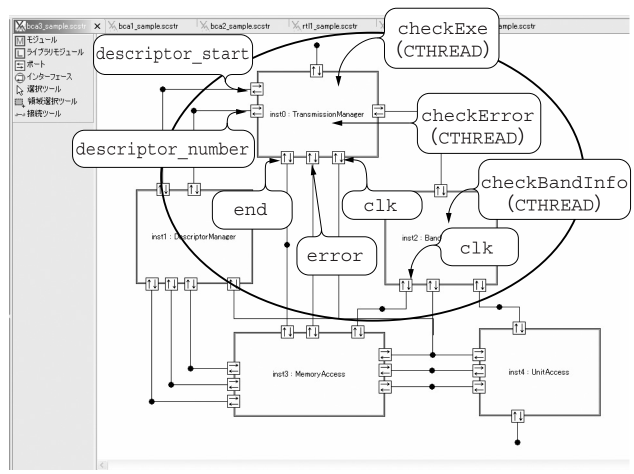
(**a**) トランザクション・レベルのモデル(ステップ1)



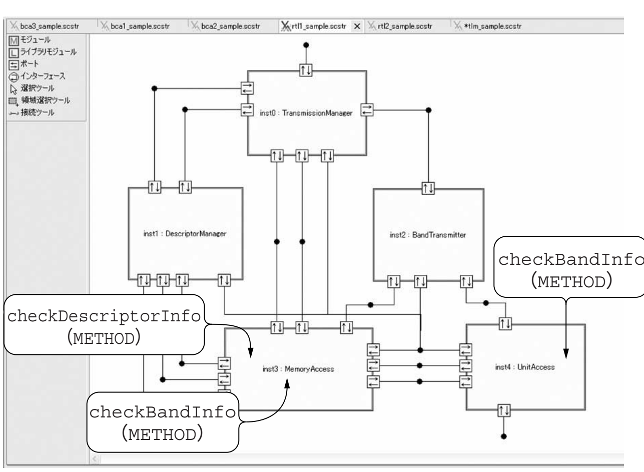
(**b**) メモリ-ユニット間通信をバス・サイクル精度に(ステップ2)



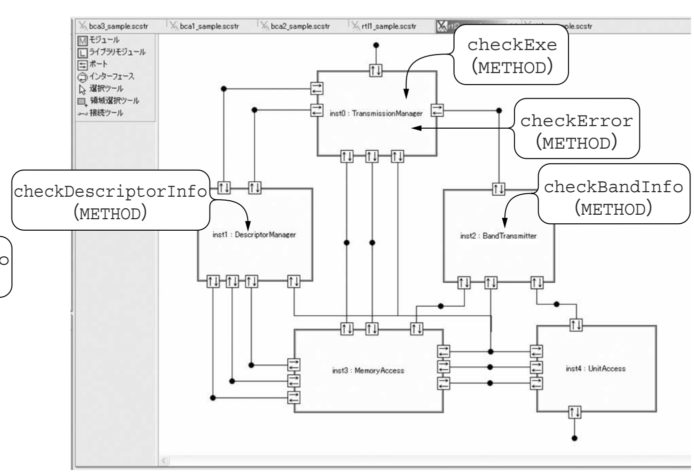
(**c**) ディスクリプタ間通信をバス・サイクル精度に(ステップ3)



(**d**) すべてをバス・サイクル精度に(ステップ4)



(**e**) メモリ-ユニット間通信をレジスタ転送レベルに(ステップ5)



(**f**) すべてをレジスタ転送レベルに(ステップ6)

図7　SystemC モデルの詳細化

(a)のトランザクション・レベル・モデルでは,プロセスの記述に THREAD プロセスを使用する.データの受け渡しには FIFO を使用する.(b),(c)は,トランザクション・レベルとバス・サイクル精度レベルが混在したモデルである.(b)ではメモリと画像処理ユニットの間を,(c)ではディスクリプタ間をクロックに同期した CTHREAD プロセスに変更する.通信についても,クロックや同期タイミング,アドレス・ビット幅,データ幅などの詳細をコーディングする.(d)では,すべてバス・サイクル精度レベルのモデルになっている.すべてをクロックに同期した CTHREAD プロセスに変更する.(e)は,バス・サイクル精度レベルとレジスタ転送レベルが混在したモデルである.メモリと画像処理ユニットの間をセンシティビティによって起動される METHOD プロセスに変更する.(f)では,すべてレジスタ転送レベルのモデルになっている.RTL 回路を意識して,すべてをセンシティビティによって起動される METHOD プロセスに変更する.

4）すべてをバス・サイクル精度に

ステップ4では，すべてをバス・サイクル精度のモデルにします（**図7（d）**）．すべてのプロセスをCTHREADプロセスにするため，すべてのモジュールにクロック（clk）が入力されます．また，TransmissionManagerモジュールとDescriptorManagerモジュールの間のインターフェースには，同期を取るためのdescriptor_startポートが追加されています．TransmissionManagerモジュールとMemoryAccessモジュールの間のインターフェースには，転送終了時の同期をとるためにendポートが追加されています．TransmissionManagerモジュール内のCheckExeプロセスでは，ディスクリプタ情報が0～3までの合計4回転送を行っています．

5）メモリ-ユニット間通信をレジスタ転送レベルに

ステップ5では，メモリから画像処理ユニットへの転送部分をレジスタ転送レベルのモデルに置き換えます（**図7（e）**）．レジスタ転送レベルは，Verilog HDLやVHDLによって論理合成ツールの入力記述を作成する場合と同じ設計抽象度です．

レジスタ転送レベルでは，フリップフロップ（レジスタ）を意識して設計する必要があります．プロセスの記述にはMETHODプロセスを使用します．一つ一つの動作はセンシティビティによって起動されます．各プロセスには，処理が終わったらそのつど終了し，再びセンシティビティを待つといった動作を記述する必要があります．

ここでは，メモリ・アクセスと画像処理ユニット・アクセスの処理についてレジスタ転送レベルの記述を行いました．すなわち，UnitAccessモジュールのcheckBandInfoプロセスとMemoryAccessモジュールのcheckBandInfoプロセスをMETHODプロセスに変更しました．これらのモジュールは，バス・サイクル精度のモデルと同じように，クロック（clk）に同期して動作します．

一方，MemoryAccessモジュールのcheckDescriptorInfoプロセスでは，アドレス（address）とサイズ（size）がセンシティビティに設定されています．つまり，checkDescriptorInfoプロセスはアドレスやサイズに変化があったときに動作することになります．

6）すべてをレジスタ転送レベルに

ステップ6では，すべてをレジスタ転送レベルの記述にします（**図7（f）**）．そのために，残りの転送管理，ディスクリプタ管理，バンド転送部分について，レジスタ転送レベルのモデルに置き換えます．すなわち，すべてのプロセスの記述をMETHODプロセスに変更します．また，DescriptorManagerモジュールのcheckDescriptorプロセスでは通信同期をとるための「descriptor_start」が，BandManagerモジュールのcheckBandInfoプロセスではバンド情報の「band_info」がセンシティビティに設定されています．さらに，TransmissionManagerモジュールのcheckErrorプロセスはエラーが生じたときだけ動作するため，エラー（error）がセンシティビティに設定されています．

● ビルドしてシミュレーションを実行

それでは実際にツールを操作してみましょう．

通常は，UMLモデリング・ツール（Rational Rose）を使って作成したクラス図をSoC Modelerに取り込み，人手で詳細化を行っていくのですが，ここでは各ステップのサンプル・モデルが用意されています．これらのモデルをツールに読み込み，モデルの動作に問題がないかどうかを確認していきます．

まず，画面上のナビゲータ領域の「tlm_sample.scstr」をダブルクリックしてトランザクション・レベルのモデル（ステップ1のモデル）を読み込みます．次に，このモデルからSystemCコードを生成します．コード生成アイコンを押すと，SystemCチェッカ・ダイアログが表示されます．このダイアログの［OK］ボタンを押すと，「SystemCジェネレーター設定」ダイアログが表示されます．保存先のディレクトリを入力して［生成］ボタンを押すと，指定したディレクトリにSystemCコードが生成されます．生成結果ダイアログのファイル名をダブルクリックすると，SystemCコードが表示されます（**図8**，この一連の動作は，この後何度も繰り返す）．

ここで，モデルの動作検証を行うためにVisual C++ .NET 2003プロジェクトを生成します．画面の［VC++ .NET2003プロジェクトを作成］ボタンを押すと作成できます．このとき，使用しているパソコンにVisual C++ .NET 2003がインストールされている必要があります．

では，動作検証を行いましょう．まず，検証ツールを起動します．Windowsのエクスプローラを開いて，上述の作業で作成したディレクトリ（例えば，c:\Work\aaa）に生成されているtlm_sample.slnというプロジェクト・ファイルをダブルクリックします．これで検証ツールが起動し

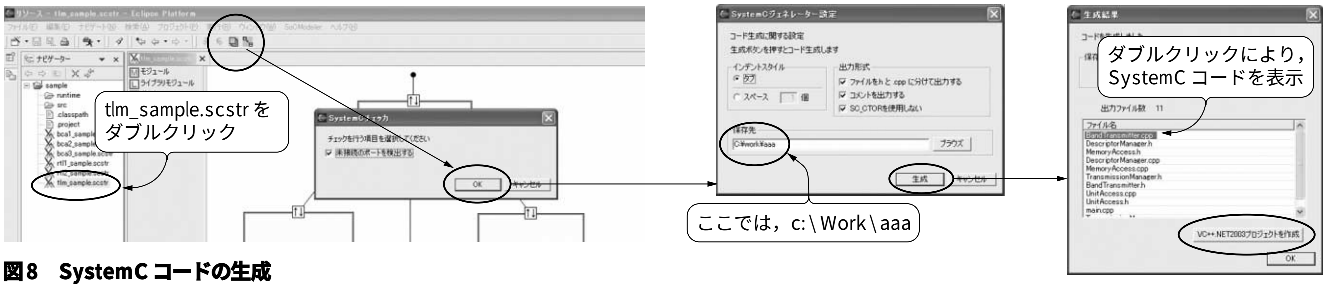


図8　SystemC コードの生成

コード生成アイコンを押すと，SystemC チェッカ・ダイアログが表示される．このダイアログの［OK］ボタンを押すと，「SystemC ジェネレーター設定」ダイアログが表示される．保存先のディレクトリを入力して［生成］ボタンを押すと，指定したディレクトリに SystemC コードが生成される．

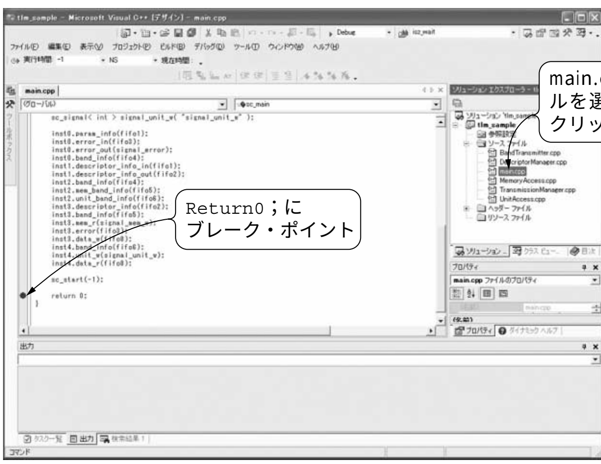


図9　ブレーク・ポイントの設定

main.cpp ファイルの最後の部分にブレーク・ポイントを設定する．シミュレーションを実行すると，設定したブレーク・ポイントの位置で停止する．



図10　シミュレーションの実行状態

シミュレーションを実行したときのコンソール画面．ソース・コードにコンソール画面表示を行うためのプログラムを記述しておけば，実行状態をコンソール画面で確認できる．

ます．検証にはSystemC Debuggerを使用しますが，このツールはVisual C++ .NET 2003のアドオン・ツールであるため，ここではVisual C++ .NET 2003が起動します．

ソリューション・エクスプローラの「ソース」の部分のファイル名をダブルクリックすると，左側のウィンドウにソース・コードが表示されます．ソース・コードの動作を確認するためにブレーク・ポイント（割り込み処理によって，プログラムの実行を停止するポイント）を設定します．ブレーク・ポイントはソース・コードの左枠の部分をマウスでクリックすることによって設定できます．今回は，シミュレーションの最後（main.cppファイルの`return 0;`の部分）にブレーク・ポイントを設定することにします（**図9**）．

では，シミュレーションを開始します．デバッグ・メニューの［開始］ボタン，または［F5］キーを押してください．ここでは，ビルド（コンパイル）がまだ行われていないため，ビルドを実行するかどうかを聞いてきます．［はい］ボタンを押すと，ビルドが実行されます．プロジェクトのファイルのうち，変更されたファイルのみをコンパイルし，ビルドが行われます．

通常は，正常にビルドが終了します．エラー・メッセージが表示されたときは，ソース・コードの記述ミスなどが考えられます．ソース・コードを修正し，もう一度ビルドを行ってください．

正常にビルドが終了したら，シミュレーションが実行されます．そして，ブレーク・ポイントの位置まで来ると，処理を停止します．今回のソース・コードには，実行状態を表示するプログラムが含まれているため，シミュレーション中の動作をコンソール画面で確認できます（**図10**）．再実行を行う（［F5］キーを押す）と最後まで実行されて終了し，コンソール画面も閉じます．

● プロセスの状態を確認する

次に，メモリから画像処理ユニットへの転送部分をバス・サイクル精度にしたモデル（ステップ2のモデル）を読み込んで，動作を確認してみます．実際の設計では，設計者自身が先ほどのトランザクション・レベルのモデルに手を加えて，このモデルを作成することになります．まず，

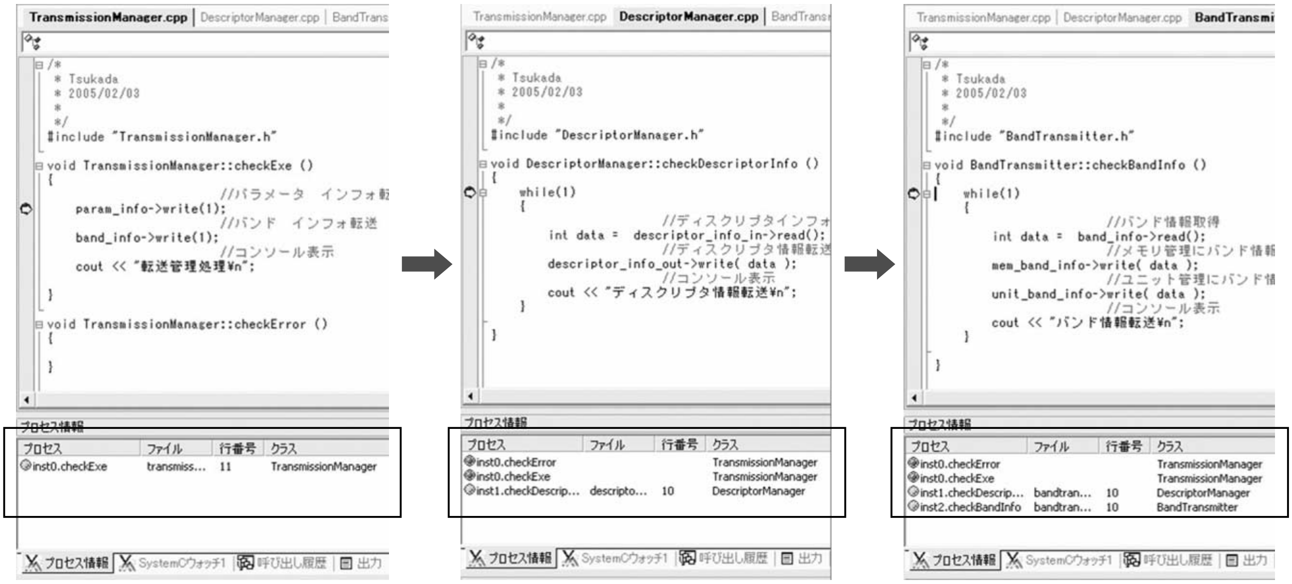


図11 プロセス状態の確認
ブレーク・ポイントで停止するたびに，プロセス状態の確認を行う．実行中プロセスは緑で，wait中プロセスは黄色で，実行終了プロセスは赤で示される．

ナビゲータ領域の「bca1_sample.scstr」をダブルクリックし，SystemCコードを生成してから，デバッガを起動してください(上述のコード生成の手順を参照)．

今度は，モデルに含まれる各プロセスの状態を確認してみましょう．BandTransmitter.cpp，DescriptorManager.cpp，MemoryAccess.cpp，TransmissionManager.cpp，UnitAccess.cppの各ファイルの先頭にブレーク・ポイントを設定してください．そして，シミュレーションを実行します．ブレーク・ポイントで停止したら，ツール・メニューの「XModelink」→「SystemCプロセス」を選択します．これで，プロセス情報を参照できます．この後，シミュレーションの再実行と停止を繰り返すことにより，プロセスの状態が変化していくようすを確認できます(**図11**)．画面上では，実行中プロセスは緑で，wait中プロセスは黄色で，実行終了プロセスは赤で示されます．

● 変数の値を確認する

ディスクリプタ部分をバス・サイクル精度にしたモデル(ステップ3のモデル)を読み込みます．先ほどと同じように，ナビゲータ領域の「bca2_sample.scstr」をダブルクリックし，SystemCコードを生成して，デバッガを起動してください．

今度は，SystemCの変数の値を確認します(SystemCウォッチ)．UnitAccess.cppなどの15行目にブレーク・ポイントを設定してください．そしてほかのブレーク・ポイントをすべて解除してください．枠の○の部分をクリックすると，ブレーク・ポイントは解除されます．そして，シミュレーションを開始します．停止したところでツール・メニューの「XModelink」→「SystemCウォッチ1」を選択す



図12 変数の値の確認
SystemCウォッチ機能を使って変数の値を確認する．ここで確認できるのは，書き込みを行った値を表示する「即値」と，デルタ遅延後に読み出した値を表示する「時間同期値」である．

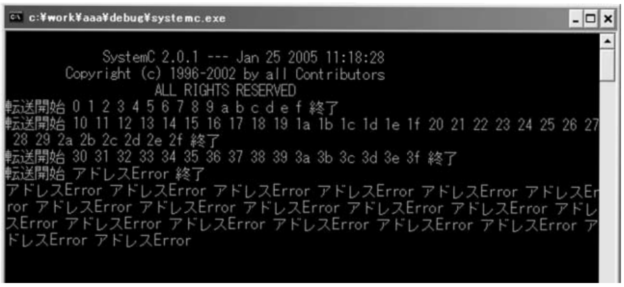


図13　転送データの確認

ソース・コードにコンソール画面表示のプログラムを記述しておけば，転送（受信）データをコンソール画面で確認することができる．



図14　トレース出力コードの生成

波形のチェックを行う．その後，［トレースファイル作成］ボタンを押して，トレース・コード・ファイルを生成する．

ると，SystemCの変数の値を参照できます（**図12**）．

ソース・コード上で参照したい変数，例えばtransfer_startをダブルクリックすると，反転した青文字の表示に変わります．その状態でマウスをドラッグし，SystemCウィンドウ上にドロップすると，現在の値が表示されます．同じようにしてdata_rとunit_wも参照できるようにしてください．シミュレーションの再実行と停止を繰り返しながら，変数の値を確認していきます．値が変化したときは，赤い文字で表示されます．

値の表示には，「即値」と「時間同期値」があります．「即値」は書き込みを行った値を表示します．「時間同期値」は読み出したときの値を表示します．動作を確認してみるとわかると思いますが，1クロックごとにデルタ遅延（シミュレーションでのみ利用される仮想的な微小遅延）が存在するため，値の書き込みを行っても，実際に書き込まれる（読み出し可能になる）のは1クロック後となります．ここでは変数の値が変化していることだけをチェックしてください．

● 画像処理ユニットの転送データをコンソール画面に表示

すべてがバス・サイクル精度になったモデル（ステップ4のモデル）を読み込みます．ナビゲータ領域の「bca3_sample.scstr」をダブルクリックし，SystemCコードを生成して，デバッガを起動してください．

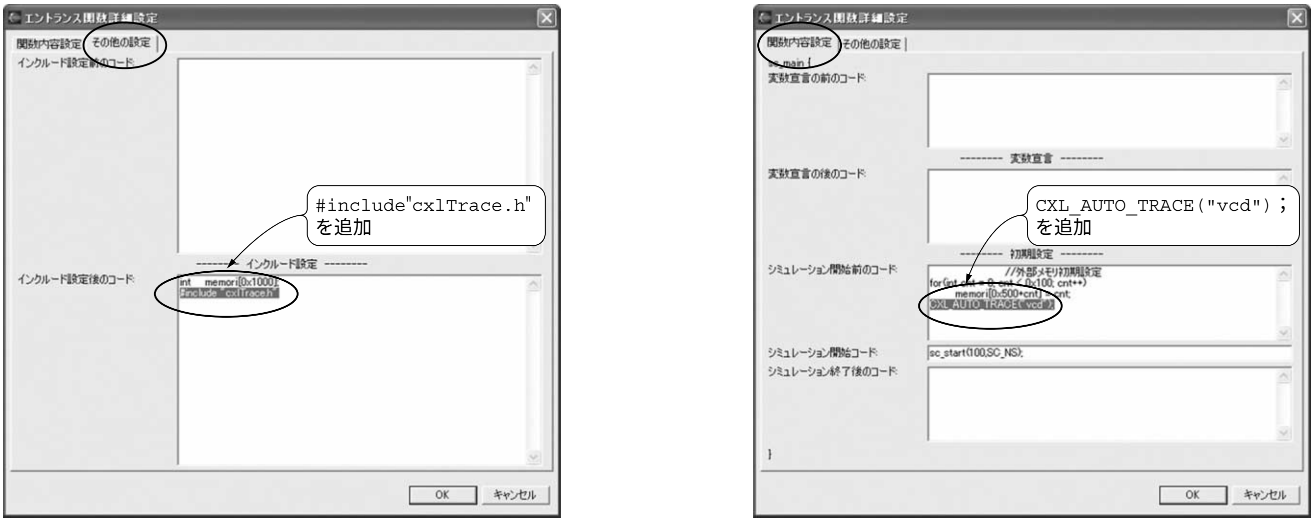
ここでは，これまで行ったコンソール画面の表示，プロセス情報の確認，変数の値の確認などをすべて試してみてください．画像処理ユニットについては，転送したデータをコンソール画面に表示するようにコーディングしてあります（**図13**）．すべてCTHREADプロセスにしているため，すべての動作がクロックに同期しています．

● RTLモデルの信号波形を観測

今度は，メモリから画像処理ユニットへのデータ転送部分をRTL記述にしたモデル（ステップ5のモデル）を読み込みます．ナビゲータ領域の「rtl1_sample.scstr」をダブルクリックし，SystemCコードを生成して，デバッガを起動してください．

ここでは信号波形を確認することにしましょう．main.cppファイルの74行目（sc_start部分）にブレーク・ポイントを設定し，シミュレーションを実行してください．停止した状態で，ツール・メニューの「XModelink」→「トレースコードの作成」を選択します．しばらくするとすべてのポートが表示されます．波形出力させたいポートのみチェックを入れて，波形出力させたくないポートのチェックをはずしてください（**図14**）．ここではunitAccess.cppのファイルをすべて表示させることにします．その後，［トレースファイル作成］ボタンを押して，ファイルを作成します．その際，作成ディレクトリはソース・ファイルの場所と同じカレント・ディレクトリとしてください．ここでいったんシミュレーションを停止し，SystemC Debuggerを終了します．

次にSoC Modelerの画面に戻り，先ほど作成したトレース・コード・ファイルのインクルードを行います．モデル・ウィンドウの部分でマウスの右クリックを行うと，ポップアップ・メニューが表示されます．このメニューの「エントランスコード編集」を選択します．「その他の設定」タブを選択し，インクルード設定後のコードに先ほど作成したファイルをインクルードするため，「#include"cxlTrace.h"」を追加入力してください．また，「関数内容設定」タブの「シミュレーション開始前のコード」の部分に「CXL_AUTO_



(a)「その他の設定」タブ (b)「関数内容設定」タブ

▲図15 エントランスコード編集

波形出力を行うためのプログラムをメイン・ファイル(sc_main)に追加する.

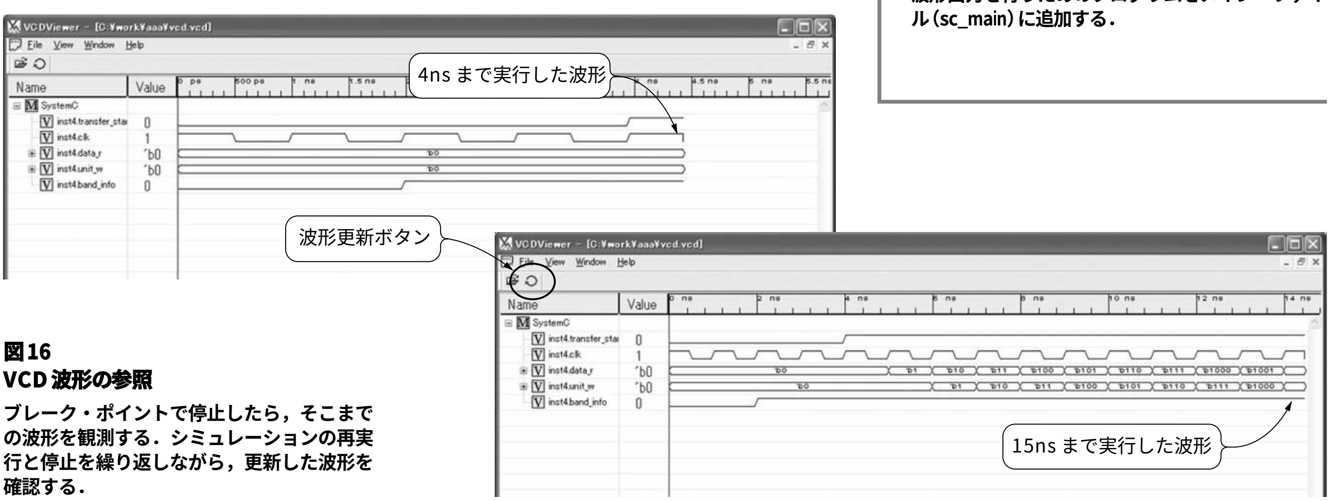


図16
VCD波形の参照

ブレーク・ポイントで停止したら,そこまでの波形を観測する.シミュレーションの再実行と停止を繰り返しながら,更新した波形を確認する.

TRACE("vcd");」を追加してください(**図15**).

再びコード生成を行い,デバッガを起動した後,シミュレーションを実行します.ビルドは自動的に行われます.ビルド・エラーが発生した場合は記述のスペルなどを見直してください.

ブレーク・ポイントを設定し,UnitAccess.cppファイルの15行目で停止します.停止した状態で,ツール・メニューの「XModelink」→「VCDビューアの起動」を選択すると,VCD(value change dump)ビューアが立ち上がります.ここでVCDビューアの中の「File」→「Open」を選択します.先ほど「CXL_AUTO_TRACE("vcd");」を指定したので,「VCD」を選択します.これにより,4ns(ナノ秒)までの信号波形が表示されます(**図16**).その後,シミュレーションの実行を10回ほど繰り返し,波形更新ボタンを押すと,15nsまでの波形が表示されます.このようにして,実行したところまでの信号波形を逐次観測できます.これらの信号波形から出力ポートの状態を確認できます.

● 時間を指定して信号波形を観測

最後に,すべてRTL記述にしたモデル(ステップ6のモデル)を読み込みます.ナビゲータ領域の「rtl2_sample.scstr」をダブルクリックし,SystemCコードを生成して,デバッガを起動してください.先ほどと同じように波形観測の設定を行いますが,今度はブレーク・ポイントまで実行するのではなく,時間を指定してシミュレーションしてみましょう.

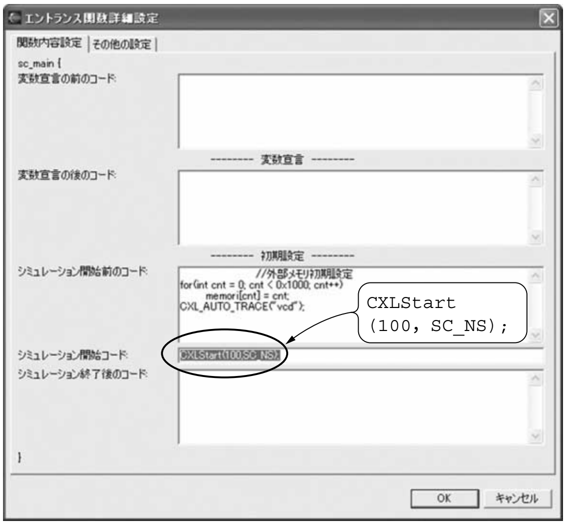




図17　時間指定実行

時間指定実行を行うために，メイン・ファイルのコードを修正する．その後，時間を指定してシミュレーションを実行する．

波形出力の設定方法は同じです．シミュレーションの開始コードは，通常，`sc_start`関数ですが，ここでは`CXLStart`関数をコールする必要があります．SoC Modelerの画面のモデル・ウィンドウ領域でマウスを右クリックし，ポップアップ・メニューを開いて，「エントランスコード編集」を選択します．そして，関数内容設定タブのシミュレーション開始コード設定部分を「`CXLStart(100,SC_NS);`」に変更し，コード生成を行い，デバッガを起動します．

ここで，これまでのようにすぐにシミュレーションを実行するのではなく，左上のメニュー部分に実行時間を入力し，returnを入力した後，[時間実行ボタン]を押します．すると，実行時間指定ウィンドウが表示されます．[上記内容でデバッガを継続する]ボタンを押すたびに，10ns分時間が進みます．これにより，VCD波形を10ns単位で観測することができます（**図17**）．

● 今後はビヘイビア合成でRTLを自動生成

今回の作業はこれで終了ですが，実際の設計ではこの後，レジスタ転送レベルのSystemCコードをVerilog HDLコードやVHDLコードに変換します．そして，論理合成ツールによってゲート・レベル回路を生成し，タイミング解析ツールなどを使ってタイミング検証を行います．

今回はフレッシャーズ向けの解説ということもあって，トップダウン設計の教科書どおり，「トランザクション・レベル→バス・サイクル精度レベル→レジスタ転送レベル」と，段階を追って詳細化していきました．しかし，実際の設計では，バス・サイクル精度レベルのコーディングを行った後，ビヘイビア合成ツールを使ってRTLコードを自動生成することが理想的です．ビヘイビア合成ツールは，通常，設計者が机上で検討しているスケジューリング（クロックやパイプライン処理の定義）やリソース・アロケーション（使用する演算器やレジスタの選択）といった作業を代行してくれます（もちろん，能力にいろいろと制約はあるが…）．こうしたツールを利用することで，設計者はアルゴリズム部分（複雑な算術演算など）の設計に注力できます．

今後は，「状態が変化していくような順序回路についてはバス・サイクル精度レベル（CTHREAD）でコーディングを行い，ビヘイビア合成ツールを使ってRTL回路を生成する」，「論理ゲートを意識する必要がある組み合わせ回路部分はレジスタ転送レベル（METHOD）でコーディングを行う」，といった使い分けがなされていくと思います．こうすることによって，システムLSIの開発効率が上がります．現在，ビヘイビア合成ツールはそれほど普及していませんが，いずれこうした状況は変わっていくだろうと筆者は考えています．

つかだ・ゆういち
キャッツ（株）
http://www.zipc.com/

<筆者プロフィール>
塚田雄一．1989年，キャッツ入社．OEM製品や自社ブランド製品（ロジックアナライザ＆ROMエミュレータ「EMUSE-G II」）の開発を担当．現在，2004年 ESEC LSIオブザイヤーグランプリを受賞した「XModelink」製品を担当．